View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by UPCommons. Portal del coneixement obert de la UPC

Universitat Politècnica de Catalunya
Departament de Llenguatges i Sistemes Informàtics
Màster en Computació

Tesi de Màster

Verifying consistency between structural and behavioral schemas in UML

Estudiant: **Elena Planas Hortal**

Director(s): **Cristina Gómez i Jordi Cabot**

Data: 25 de juny, 2008

UNIVERSITAT POLITÈCNICA DE CATALUNYA
Departament de Llenguatges i Sistemes Informàtics
Master in Computing

Title of thesis : Verifying consistency between structural and behavioral schemas in UML
Student : Elena Planas Hortal
Advisor : Cristina Gómez *Co-advisor* : Jordi Cabot
Date : June 25th, 2008

Abstract :

The specification of an information system must include all relevant static and dynamic aspects of the domain. The static aspects are collected in structural diagrams that are represented in UML by means of class diagrams. Dynamic aspects are usually specified by means of a behavioral schema consisting of a set of system operations (composed by actions) that the user may execute to query and/or modify the information modeled in the class diagram.

Behavioral schemas must be consistent with regard to structural schemas. Consistency between both schemas means that the set of system operations provided by designers must be *syntactically consistent* (i.e, the operation specifications conform to a particular syntax), *executable* (i.e, for each operation there must exist a system state over which the operation can be successfully applied), *complete* (i.e, through these operations, users should be able to modify the population of all modifiable elements in the class diagram) and *non-redundant* (i.e, there are not (partly) superfluous operations).

The goal of this thesis is to give a method to determine the consistency between structural and behavioral schemas of an information system. Moreover, in case of inconsistent schemas the method must provide feedback information to allow designers modify their behavioral schemas in order to repair the inconsistency.

Keywords : verification, UML, structural schema, behavioral schema, Action Semantics

Language : English

Contents

1. Introduction 7
1.1. Objectives of this master thesis 8
1.2. Document structure 9

2. Basic Concepts 11
2.1. Structural Schema 11
2.2. Behavioral Schema 12
2.3. Action Semantics in the UML 13
2.3.1. Action Semantics syntax proposed 14
2.4. Dependencies among actions 18

3. The Proposed Method 21
3.1. Overview 21
3.2. Analyzing Syntactic Consistency 22
3.3. Determining Execution Paths 25
3.4. Verifying Weak Executability 27
3.4.1. Executability algorithm 29
3.5. Verifying Completeness 36
3.5.1. Completeness algorithm 36
3.6. Detecting Redundant Paths 39
3.6.1. Redundancy in Actions 40
3.6.2. Redundancy in Execution Paths 41
3.6.3. Redundancy in Operations 41

4. Related Work 43

5. Conclusions and Further Work 45

6. References 47

1. Introduction

Since the very beginning of computer science, one of the main goals of software engineers has been to automate as much as possible the software development process. In fact, the software engineering community envisages a future in which, of all the phases of software development, software engineers will only be strictly necessary during the specification of the information system while the remaining phases (mainly design, implementation and test) would be fully automated. This is one of the most challenging and long-standing goals in software engineering [34].

This is also the focus of some of the most popular and current development approaches as MDD (Model-Driven Development [2]) and MDA (Model-Driven Architecture [26]). MDD is a software engineering paradigm that gives the Conceptual Schema (CS) (that is, a representation of knowledge about a domain) a central role in the development process and promote the automatic generation of the system implementation based on its CS, either directly or by first transforming the CS into a new model adapted to the specific features and characteristics of the target platform. MDA is the OMG vision of MDD and it is founded on standards like MOF and OCL for modelling and meta-modelling. In MDA approach there are two kinds of models: PIM (Platform-Independent Model), that provide formal specification of the structure and behavior of the system away technical details (for example, a CS is a PIM), and PSM (Platform Specific Model), that specify the system in terms of the implementation constructs that are available in one specific implementation technology.

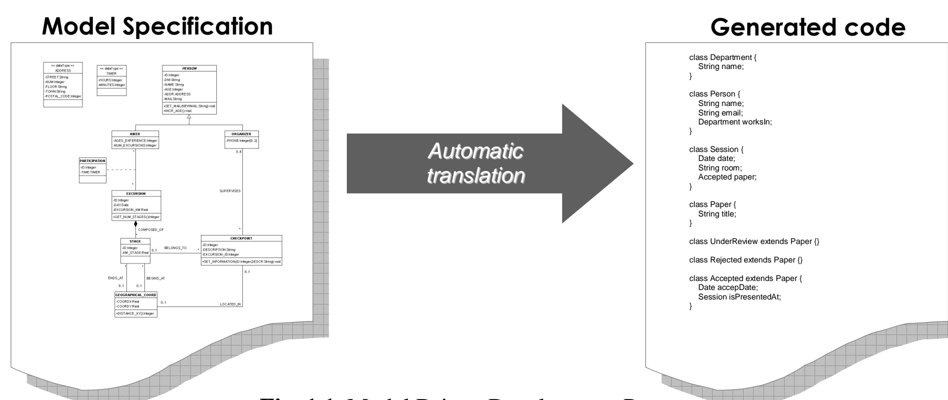


Fig. 1.1. Model Driven Development Process.

CS is a key artifact in development process and, thus, its correctness is essential. Wrong CSs can lead to incorrect implementations. Here is where the *verification methods* come into play. There are many methods for verifying CSs. The verification depends on the type of the model and on the property we want to verify. Most of the current methods are focused on the static part of the CS but less work has been done with respect to the dynamic part (behavior). Verification of the correctness properties of the both CS parts is very important to guarantee the quality of application that will be obtained from the model.

In particular, we are interested in the verification of Action Semantics (AS) since it is a key element in all executable UML methods [25, 17] to specify the behavior of the operations defined in the class diagram. Actions are the fundamental unit of behavior specification. Basic actions include the creation of new objects, creation of new links, removals of existing objects or the modification of attribute values, among others, and they can be coordinated with conditional and loop nodes to completely define the operation effect.

As a simple example, consider the class diagram of Fig. 1.2, described by UML language [39]. The class diagram describes the objects within a system (*people* and *departments*) and their relationships (*person works in a department*). Fig. 1.2 also includes the behavior of the system through the operations specified by AS (*addPerson* and *changeAddress*). In this context, both operations are incorrect, since *changeAddress* tries to update an attribute (*address*) which does not even exist in the diagram and *addPerson* can never be successfully executed (i.e. every time we try to execute *addPerson* the new system state violates the minimum '1' cardinality constraint of the department role in *WorksIn* since the new person instance is not linked to any department). Besides, this operation set is not complete, that is, through these operations users cannot modify all elements of the class diagram, e.g. it is not possible to create and destroy departments. If these errors are not fixed before continuing with the code-generation phase, the resulting system implementation will be totally useless.

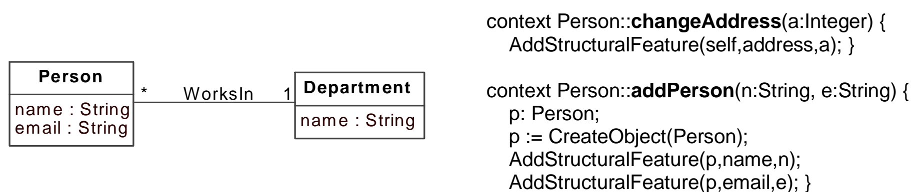


Fig. 1.2. A simple example of a class diagram with two operations.

In this sense, the goal of this master thesis is to provide a set of techniques for the verification of correctness properties of action-based behavior specifications at design time. The correctness properties that we deal with can be summarized in:

- **Syntactic consistency**: The operation specifications conform to a particular syntax, in this case, the UML metamodel [30] syntax and its restrictions.
- **Executability**: The execution of operations leaves the system in a consistent state.
- **Completeness**: All possible changes on the system state can be performed through the execution of operations.
- **Redundancy**: There are not (partly) superfluous operations.

1.1. Objectives of this master thesis

The objective of this master thesis is to complement the current verifying methods of dynamic part of a system with a new method able to verify the previous set of correctness properties.

This objective can be divided into the following functional goals:

1. Analyze the *syntactic consistency* of each operation defined in the CS.
2. Determine the *executability* of each operation defined in the CS, taking into account the dependencies among actions.

3. Determine the *completeness* of the whole operation set defined in CS.
4. Detect possible *redundancies* in operations defined in the CS.
5. For each detected error, suggest to the designer possible corrective procedures as a complementary *feedback*.

To achieve the goals 2 and 3, it is necessary to consider a new sub-goal consisting in the analysis of the operation flow to determine all possible execution paths in an operation, that is, a sequence of actions that may be followed during the operation execution.

Additionally, we are interested in the following non-functional goals:

1. Perform a static analysis (do not a model animation/simulation during the verification process).
2. Do not reduce the language expressiveness.
3. Be complete, in the sense that the existence of a solution can always be determined.

1.2. Document structure

This master thesis is structured as follows. The next section introduces basic concepts that are important to understand the rest of the work. Section 3 give an overview of the method proposed and describe its steps in detail. Section 4 presents the state of the art of CS verification methods. Finally, in section 5 we present the conclusions and indicate areas of future work.

2. Basic Concepts

This section describes the main concepts that are important to understand the rest of the work.

2.1. Structural Schema

A Conceptual Schema (CS) is a representation of general knowledge about a domain [33]. A CS includes two main components: Structural Schema and Behavioral Schema.

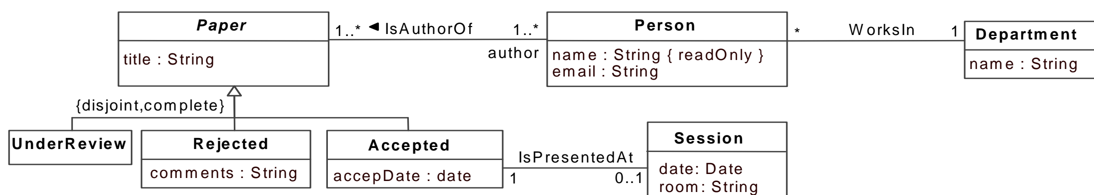
A structural schema specifies the static part of a system, that is, the representation of a problem domain. A structural schema can be represented with a UML Class Diagram (CD) artifact, that contains information about the domain classes, attributes, associations, generalizations and constraints.

We consider that a CD can be represented using the tuple:

$$CD = \langle set_{cl}, set_{at}, set_{as}, set_{gen}, set_{constr} \rangle$$

where set_{cl}, set_{at}, set_{as}, set_{gen} and set_{constr} represent the set of classes, attributes, associations, generalization sets and constraints of the class diagram *CD*, respectively. All elements in *CD* are assumed to be correct instances of the corresponding metaclasses of the UML metamodel [30]. We assume that all associations are binary associations (n-ary associations can be easily expressed in terms of a set of binary ones [24]).

As an example, the Fig. 2.1.1 shows a class diagram aimed at representing part of a conference management system. The abstract class *Paper* is specialized in three disjoint subclasses according to the state of the paper (*UnderReview*, *Rejected* or *Accepted*). Only accepted papers can be presented in a *Session*. Each paper is written by one or more people, each of which works in a *department*. The constraint *MaxPapersSent* means that a person may send at most 10 papers.



context Department **inv** MaxPapersSent:
self.person.paper → asSet() → size() <= 10

Fig. 2.1.1. Example of a class diagram.

For this class diagram we have that:

set_{cl} = { Paper, UnderReview, Accepted, Rejected, Person, Department, Session }

set_{at} = { title, accepDate, comments, name, email, name, date, room }

set_{as} = { IsAuthorOf, WorksIn, IsPresentedAt }

set_{gen} = { Paper, {UnderReview, Accepted, Rejected} }

set_{constr} = { context Department inv MaxPapersSent: self.person.paper → asSet() → size() <= 10 }

2.2. Behavioral Schema

A behavioral schema specifies the dynamic part of the system, that is, the functionalities that a system can perform. In UML there are many ways to represent the behavior of a system (Sequence Diagram, State Machines, etc.) but the basic way is the use of operations (attached to classes) that the user may execute to query and/or modify the information modeled in the structural schema.

The operations of a behavioral schema can be defined by two ways: declarative or imperative approaches.

In a declarative specification, a contract for each operation must be provided [27]. A contract consists of a set of preconditions and postconditions. The precondition expresses requirements that any call must satisfy if it is to be correct and the postcondition expresses properties that are ensured in return by the execution of the call. The contracts may be represented by OCL language [31], a declarative language provided by OMG (Object Management Group).

In an imperative specification, the conceptual modeler explicitly defines the set of actions (insertion of a new object, update of an attribute,…) to be applied over the system state. Imperative specifications of operations may be defined by an imperative language as UML Action Semantics (AS) [32].

In conceptual modeling, the declarative approach is preferable since it allows a more abstract and concise definition of the operation effect and conceals all implementation issues. Nevertheless, in order to transform a model specification to a set of executable software components, declarative specifications must be transformed into their equivalent imperative ones. This transformation is non-deterministic, but an initial research [6] has been done to provide some heuristics for helping in the translation process. In this thesis we assume that the starting point is the use of imperative specifications of operations.

As an example, we have defined four operations for the class diagram of Fig. 2.1.1. Its signature is:

- *endOfReview(com:String, d:Date, evaluation:String)*: Reclassifies a paper as rejected or accepted depending on the evaluation parameter.
- *submitPaper(tit:String, authors:Person[1..*])*: Creates a new under review paper and links the paper with its authors.
- *dismiss()*: Deletes the WorksIn link between a person and his/her department.
- *createSchedule(dateList: List(Date), roomList: List(String))*: Assigns a date and room to present each accepted paper.

2.3. Action Semantics in the UML

In UML, the behavior of an operation can be specified using several UML constructs as State Machines or Activities, among others (see the UML metamodel fragment of Fig. 2.3.1). In this thesis we will concentrate on this latter option.

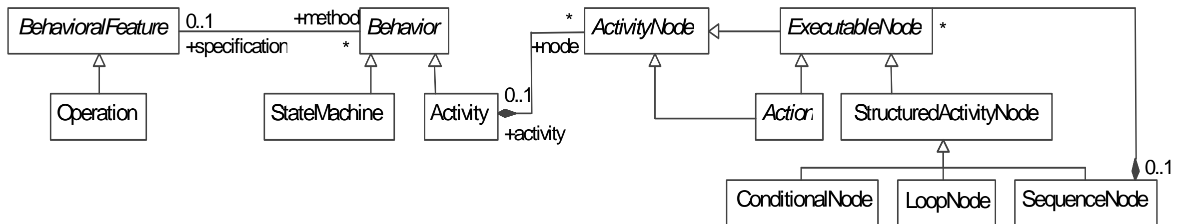


Fig. 2.3.1. Fragment of UML metamodel.

Activities describe a procedural implementation of the operation effect (in contrast with declarative definitions based on the use of pre and postcondition). An activity is composed of a set of activity nodes describing the different steps of the activity. Activity nodes may either be basic actions or structured nodes useful to coordinate basic actions in action sequences, conditional blocks or loops. We consider two types of loops: *while-do* (if meta-property *isTestedFirst* from metaclass *LoopNode* is equal to true, that is, the test is performed before the first execution of the body) and *do-while* (if meta-property *isTestedFirst* is equal to false, that is, the body is executed once before the test is performed).

UML metamodel defines a set of actions that allows to specify the behavior of an operation. In full, UML describes more than forty actions, but for the purposes of this thesis we only consider a subset of them. Specifically, we will focus on the construction actions (*CreateObjectAction*, *CreateLinkAction*), destruction actions (*DestroyObjectAction*, *DestroyLinkAction*) and update actions (*AddStructuralFeatureValueAction*, *ReclassifyObjectAction*).

UML provides an abstract syntax for these actions [30], but it does not define a concrete syntax for their use in an operation specification. Therefore, it is necessary to define a concrete syntax in order to complete the abstract specification. There are several non-standard proposals that define an abstract and concrete syntax for actions. Some of these proposals are: Action Semantics Language (ASL) [8], Object Action Language (OAL) [29], Platform-independent Action Language (PAL) [37], PathMATE Action Language (PMAL) [36], Starr's Concise Relational Action Language (SCRALL) [41], Shlaer-Mellor Action Language (SMALL) [40] and That Action Language (TALL) [25].

For the sake of simplicity and according to the goals of this thesis, we use our own syntax, according to the abstract syntax provided by UML standard action metaclasses and similar to the previous proposals. Details of this syntax are explained in next section.

2.3.1. Action Semantics syntax proposed

In the following, we describe the concrete syntax of actions used in this work. For each action we specify:

- *Abstract syntax*: UML abstract syntax.
- *UML metamodel*: extract of UML metamodel for the action.
- *Concrete syntax*: our concrete syntax.
- *Arguments*: input arguments of the action.
- *Result*: output arguments of the action.
- *Basic semantics*: basic semantics defined by UML.
- *Additional Semantics*: additional semantics added.

Abstract syntax	**CreateObjectAction**
UML metamodel	
Concrete syntax	**CreateObject(class:Classifier): InstanceSpecification**
Arguments	*class* – Classifier to be instantiated.
Result	Returns the object instantiated.
Basic semantics	This action creates a new object that conforms to the specified classifier. The specified classifier cannot be abstract. The action has no other side effects. In particular, the new object has no structural feature values and participates in no links.
Additional semantics	-

Abstract syntax	**DestroyObjectAction**
UML metamodel	
Concrete syntax	**DestroyObject(o:InstanceSpecification)**
Arguments	*o* – Object to be destroyed.
Result	-
Basic semantics	This action destroys the object *o*. Destroying an object that is already destroyed has no effect.
Additional semantics	We assume that links in which *o* participates are not automatically destroyed (attribute *isDestroyLinks = false*). We restrict the use of this action only for destroy objects that are instances of a class (not for destroy links).

Abstract syntax	**AddStructuralFeatureValueAction**
UML metamodel	
Concrete syntax	**AddStructuralFeature(o:InstanceSpecification, at:StructuralFeature, v:ValueSpecification)**
Arguments	*o* – Object to be updated. *at* – Attribute to be updated. *v* – New value.
Result	-
Basic semantics	This action sets the value *v* as the new value for the attribute *at* of the object *o*.
Additional semantics	We assume that multi-valued attributes are expressed (and analyzed) as binary associations between the class and the attribute data type. For this reason, the new value always is inserted in first position. We restrict the use of this action to the addition of values to structural features, it cannot be used for modifying association ends.

Abstract syntax	**CreateLinkAction**
UML metamodel	
Concrete syntax	**CreateLink(as:Classifier, p1:InstanceSpecification, p2:InstanceSpecification)**
Arguments	*as* – Link to be created. *p1, p2* – Participants of the link.
Result	-
Basic semantics	This action creates a new link in the binary association *as* between objects *p1* and *p2*. The association cannot be an abstract classifier. Rcreating an existing link has no effect if the structural feature is unordered and non-unique.
Additional semantics	-

Abstract syntax	**DestroyLinkAction**
UML metamodel	
Concrete syntax	**DestroyLink(as:Classifier, p1:InstanceSpecification, p2:InstanceSpecification)**
Arguments	*as* – Link to be destroyed. *p1, p2* – Participants of the link.
Result	-
Basic semantics	This action destroys the link between objects *p1* and *p2* from *as*. There is no return value in either case. Destroying a link that does not exist has no effect.
Additional semantics	-

Abstract syntax	**ReclassifyObjectAction**
UML metamodel	Action (fromBasicActions); ReclassifyObjectAction — isReplaceAll : Boolean = false; +object {subsets input} 1 → InputPin (from BasicActi...); +oldClassifier * → Classifier (fromKernel); +newClassifier * → Classifier (fromKernel)
Concrete syntax	**ReclassifyObject(o:InstanceSpecification, newClass:Classifier[0..*], oldClass: Classifier[0..*])**
Arguments	*o* – Object to be reclassified. *newClass* – New superclasses of *o*. *oldClass* – Old superclasses of *o*.
Result	-
Basic semantics	This action adds *o* as a new instance of classes in *newClass* and removes it from classes in *oldClass*. Multiple classifiers may be added and removed at a time. None of the new classifiers may be abstract.
Additional semantics	-

According to the previous concrete syntax, the imperative specification of the operations introduced in section 2.2 is:

```
context Paper::endOfReview(com:String, d:Date,
evaluation:String) {
  if self.oclIsTypeOf(UnderReview) then
    if evaluation = 'reject' then
      ReclassifyObject(self,Rejected,∅);
      AddStructuralFeature(self,comments,com);
    else
      ReclassifyObject(self,Accepted,∅);
    AddStructuralFeature(self,accepDate,d);
    endif
  endif
}
```

```
context Paper::submitPaper(tit:String, authors:Person[1..*]) {
   i: Integer := 1;
   p := CreateObject(UnderReview);
   AddStructuralFeature(p,title,tit);
   while i ≤ authors->size() do
      CreateLink(IsAuthorOf,p,authors[i]);
      i := i+1;
   endwhile
}
```

```
context Person::dismiss() {
  DestroyLink(WorksIn,self,self.department);
}
```

```
context Paper::createSchedule(dateList: List(Date),
roomList: List(String)) {
     acceptPapers: List(Accepted);
     acceptPapers := (Accepted.allInstances());
     s: List(Session);
     i: Integer := 1;
     while i ≤ acceptPapers->size() do
       s[i] := CreateObject(Session);
       AddStructuralFeature(s[i],date,dateList[i]);
       AddStructuralFeature(s[i],room,roomList[i]);
       i := i+1;
     endwhile
     CreateLink(isPresentedAt,acceptPapers[1],s[1]);
  }
```

Fig. 2.3.2. Specification of *endOfReview*, *submitPaper*, *dismiss* and *createSchedule* operations.

In the next sections we will show how to verify these operations.

2.4. Dependencies among actions

A dependency from an action $action_1$ (the depender action) to an action $action_2$ (the dependee action) expresses that $action_2$ must be included in all operations where $action_1$ appears to avoid violating the constraints of the class diagram. It may happen that $action_1$ depends on several actions (AND-composition). In this case, all dependee actions must be included in order to satisfy the dependency. It may happen also that $action_1$ has different alternatives to keep the consistency of the system OR-composition). In this case, as long as one of the possible dependee actions appears in the operation, the dependency is satisfied. As we will show in next sections, this concept is used for verifying the executability of operations.

Dependencies between actions depend on the type of the action and on the integrity constraints of each particular class diagram. For the purposes of our analysis, we just need to consider minimum cardinality constraints for associations and disjoint and complete constraints for generalizations (either graphically represented or implicitly induced by textual OCL constraints).

As a simple example, consider the class diagram of Fig. 2.1.1. In this context, if we have an operation that includes an action *p:=CreateObject(Person)*, this operation requires the presence of the action *CreateLink(WorksIn,p,d)*, where *d* is a department, in order to satisfy the minimum '1' cardinality constraint of the department role in *WorksIn* association. If we have another operation that includes an action *Reclassify(p,{Accepted},{})* to reclassify a paper *p* from *UnderReview* to *Accepted*, this operation must contain the action *Reclassify(p,{},{UnderReview})* in order to satisfy the disjoint constraint of the generalization.

For other types of constraints, we can always find a combination of a system state and/or a set of arguments for which the execution of an action *action* results in a consistent state with respect to those constraints. For instance, maximum cardinality constraints are never violated when action is applied to an empty system state. Constraints restricting the value of the attributes of an object may be satisfied when passing the appropriate arguments as parameters for the action. As an example, the *MaxPapersSent* constraint (Fig. 2.1.1) restricts the possible values of the parameter authors but we can always find a person (e.g. a newly created one) that may submit a paper without violating this constraint. The same situation occurs with constraints restricting the relationship between an object and related objects. Therefore, all these constraints are ignored when computing the dependencies of action.

A simple dependency for an action *action* is defined as *action*→*dep* where *dep* is the action required by action. Complex dependencies are expressed as a sequence of simple ones joined with the logical AND and OR operators.

The following table provides the rules to compute the dependencies (second column) required by each action type (first column). These rules are adapted from [7] where they were expressed using a proprietary list of action types. Note that most of the rules include applicability conditions that precise in more detail when the dependency is required.

Table 2.4.1. Dependencies for modification actions. *Min(c_i,as)* and *max(c_i,as)* denote the minimum (maximum) multiplicity of c_i in *as* (for reflexive associations we use the role name).

Action	Required Actions
o=CreateObject(c)	*AddStructuralFeature(o,at,v)* for each non-derived and mandatory attribute at of *c* or of a superclass of *c*
	AND <*min(c,as),CreateLink(as,o,o_2)*> dependencies for each non-derived association as where *c* or a superclass of *c* has mandatory participation
DestroyObject(o:c)	<min(c,as),*DestroyLink(as,o,o_2)*> for each non-derived as where *c* or a superclass of *c* has a mandatory participation
CreateLink(as,o_1:c_1,o_2:c_2) (when *min(c_1,as) = max(c_1,as)*) to be repeated for the other end	*DestroyLink(as,o_1,o_3)* (if *min(c_2,as)* <> *max(c_2,as)*)
	OR *CreateObject(o_1)*
	OR *ReclassifyObject(o_1,c_1,Ø)*
DestroyLink(as,o_1:c_1,o_2:c_2) (when *min(c_1,as) = max(c_1,as)*) to be repeated for the other end	*CreateLink(as,o_1,o_3)* (if *min(c_2,as)* <> *max(c_2,as)*)
	OR *DestroyObject(o_1)*
	OR *ReclassifyObject(o_1,Ø,c_1)*
AddStructuralFeature(o,at,v)	-
ReclassifyObject(o,{nc},{oc})	*AddStructuralFeature(o,at,v)* for each non-derived and mandatory attribute at of each class *c* ∈ *nc*
	AND <*min(c,as),CreateLink(as,o,o_3)*> for each *c* ∈ *nc* and for each non-derived association *as* where *c* has a mandatory participation
	AND *ReclassifyObject(o,Ø,{cc'})* for a *cc'* such that ∃*cc* ∈ *nc* and *cc'* ∉ *oc* and *cc* ≠ *cc'* and *cc* and *cc'* are subclasses of a common disjoint and complete generalization set and *o* was instance of *cc'*
	AND <*min(c,as),DestroyLink(as,o,o_3)*> for each class *c* in *oc* and for each non-derived association *as* where *c* has a mandatory participation
	AND *ReclassifyObject(o,{cc'},Ø)* for a *cc'* such that *cc'* ∉ *nc* and ∃*cc* ∈ *oc* and *cc* ≠ *cc'* and *cc* and *cc'* are subclasses of a common disjoint and complete generalization set and *o* was instance of *cc*

As an example, consider the class diagram of Fig. 2.1.1 and suppose an operation that includes an action *CreateLink(WorksIn,p,d)* where p and d are a person and a department respectively.

This action has the follow dependencies:

CreateLink(WorksIn,p,d) → *DestroyLink(WorksIn,p,d') OR CreateObject(p)*

Where:

- The dependency *DestroyLink(WorksIn,p,d')* means that if we create a new link between a person *p* and department *d,* we should destroy the current link between the person *p* and his current department (*d'*).
- The dependency *CreateObject(p)* means that if we create a new link between a person *p* and department *d*, we should create the object *p*.

As we have seen, as long as one of the previous dependee actions appears in the same sequence of action *CreateLink(WorksIn,p.d)*, the dependency is satisfied.

3. The Proposed Method

This section provides an overall vision of our proposal for verifying operations specified with AS and describes in detail each step of the method.

3.1. Overview

The method we propose consists of a set of techniques for the verification of correctness properties of action-based behavior specifications at design time. Without loss of generality, we will focus on the verification of actions appearing in the definition of the operations' behaviors. However, many of our techniques could be equally used to verify action sequences appearing in other kinds of behavior specifications as State Machines or Sequence Diagrams.

Roughly, given an operation op, our method proceeds by follow a set of steps that can be summarized in (see Fig. 3.1.1):

1. Analyzing the syntactic consistency of each action *action* ∈ *op*, that is, guarantee that its specification conforms to the abstract syntax specified by the UML metamodel.
2. Analyzing the operation flow to determine all possible execution paths in *op* (an execution path is a sequence of actions that may be followed during the operation execution).
3. Determining the executability of each execution path *p* by performing a static analysis of the dependencies among the modification actions (i.e. actions that change the system state) in *p* and their relationship with the structural constraints (as cardinality constraints) in the class diagram.
4. Determining the completeness of the whole operation set, that is, all possible changes on the system state can be performed through the execution of operations.
5. Analyzing possible redundancies in *op*, that is, detecting if there are superfluous actions.

For each detected error, possible corrective procedures are suggested to the designer as a complementary feedback.

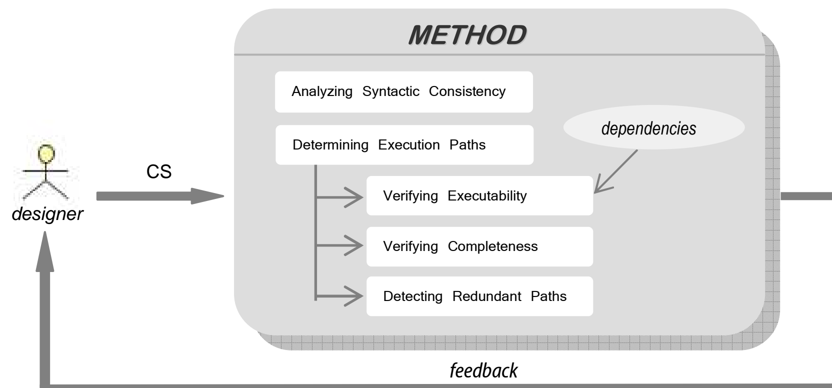


Fig. 3.1.1. Method overview.

3.2. Analyzing Syntactic Consistency

The first step of our method consists in checking the syntactic consistency of each action in an operation.

Concerning UML models, syntactical consistency ensures that a specification conforms to the abstract syntax specified by the UML metamodel [15] (similar to the grammar of programming languages). For instance, in our running example, the action *p:=CreateObject(Paper)* is not syntactically correct due to we cannot create an instance of an abstract class.

Syntactical consistency conditions are expressed in UML metamodel using a set of (OCL) constraints (i.e. Well-Formedness Rules) that restrict the possible set of valid (i.e. Well-Formed) UML models. These Well-Formedness Rules (WFRs) help to prevent syntactic errors in action specifications.

An operation is ***syntactically correct*** *when each action in the operation satisfies all these WFRs.*

In the follow we show some WFR included in the UML 2.0 metamodel [30]. The complete set of WFR can be found in the UML 2.0 metamodel.

#	**WFR 1**
UML metaclass	LinkAction (super-class of CreateLinkAction)
Textual description	When specifying a CreateLink action action over an association assoc, the type and number of the input objects (parameters) in action are compatible with the set of association ends defined for assoc.
OCL definition	context LinkAction inv: self.endData->collect(end)=self.association()->collect(connection))

#	**WFR 2**
UML metaclass	CreateObjectAction
Textual description	The classifier cannot be abstract.
OCL definition	context CreateObjectAction inv: not (self.classifier.isAbstract = true)

#	**WFR 3**
UML metaclass	ReclassifyObjectAction
Textual description	The newClassifiers set cannot contain any abstract classifier.
OCL definition	context ReclassifyObjectAction inv: not self.newClassifier->exists(isAbstract = true)

Unfortunately, our analysis of the WFRs relevant to the Action Packages has detected several flaws that compromise the usefulness of such rules (a previous analysis [3] already highlighted that other parts of the UML specification should be reviewed as well). Some example errors are the following:

- **Syntactic errors**: References to "*forall*" (instead of "*forAll*") and "*oclisKindOf*" (for "*oclIsKindOf*") operations.
- **UML 1.5 related errors**: WFRs restricting the multiplicity of *input* and *output* pins refers to a multiplicity attribute that does not longer exist in the UML metamodel (in UML 2.0, pins are subtypes of *MultiplicityElement* and thus we should use the *upper* and *lower* attributes instead). There is also a reference to the now inexistent *NavigableEnd* metaclass.
- **Semantic errors**: The constraint "*context WriteStructuralFeatureAction inv: self.value.type = self.structuralFeature.**featuringClassifier***" forces the type of the new value for the structural feature to be equal to the type of the Classifier owning the feature. Clearly, this is plain wrong. The type of the new value should be the same as the type of the structural feature, i.e. "*self.value.type = self.structuralFeature.**type***".
- It is not clear the relationship between the *InstanceSpecification*, *ValueSpecification* and *Pin* metaclasses. Since input and output pins must hold *InstanceSpecification* (e.g. in the *CreateObjectAction* action) and *ValueSpecification* (e.g. *WriteStructuralFeature* actions) values, both kind of values need to be converted to instances of the *Pin* metaclass which is not possible with the current metamodel structure.

Besides, several required WFRs are not predefined in the metamodel, and thus, existing WFRs must be complemented with new ones to guarantee that action specifications are syntactically correct.

For instance, the rules that we propose to add in UML metamodel are:

#	**WFR4**
UML metaclass	WriteStructuralFeatureAction
Textual description	The type of the input object (i.e. the object whose feature will be modified) is compatible with the classifier owning the feature.
OCL definition	context WriteStructuralFeatureAction inv: self.value.type = self.structuralFeature.type

#	**WFR5**
UML metaclass	CreateObjectAction
Textual description	The input classifier cannot be the supertype of a covering generalization set (in a covering generalization, instances of the supertype cannot be directly created).
OCL definition	context CreateObjectAction inv: Generalization.allInstances()->exists(g|g.general = self) implies c.generalizationSet.isCovering = false

#	**WFR6**
UML metaclass	ReclassifyObjectAction
Textual description	The newClassifiers set and the oldClassifiers set are disjoint sets.
OCL definition	context ReclassifyObjectAction inv: self.oldClassifier->excludesAll(self.newClassifier)

#	**WFR7**
UML metaclass	ReclassifyObjectAction
Textual description	The newClassifiers set cannot contain the supertype of a covering generalization set.
OCL definition	context ReclassifyObjectAction inv: let: superclasses = Generalization.allInstances().general in: self.newClassifier->forAll(c | superclasses->includes(c) implies c.type.powertypeExtent.isCovering = false)

#	**WFR8**
UML metaclass	WriteStructuralFeatureAction
Textual description	Values of readOnly attributes are not updated after their initial value has been assigned.
OCL definition	This rule cannot be expressed in OCL.

Once the previous ill-defined WFRs are fixed and the new ones are added to the metamodel specification, we can successfully analyze the well-formedness of action specifications as a first step of our verification process. The basic idea is to apply an algorithm that, for each action, the algorithm checks the previous WFR.

The graphical overview of this step would be:

Fig. 3.2.1. Analyzing Syntactic Consistency overview.

The input of the step is the CS, in particular the imperative specification of each operation *op* included in the CS. For each *action* $\in$ *op*, the method would check the previous WFR and would return as a feedback the violated rules and the actions that violate them.

As an illustrative example of this step, suppose that we have a new operation in our running example (Fig. 2.1.1):

```
context Person::setEmail(e: Integer) {
  AddStructuralFeature(self,email,e);
}
```

If we apply the previous WFR to the set of actions of the operation *setEmail* (in this case, a unique action *AddStructuralFeature(self,email,e))* we detect that the rule WFR4 fails, due to the type of the input object (*Integer*) is incompatible with the type of the attribute *email* (*String*).

After this first analysis, we are ready to proceed with a more semantic verification process involving a more complex analysis of the relationship between the actions specified in operations and the elements of the class diagram.

3.3. Determining Execution Paths

The correctness properties that will be presented in the next sections are based on an analysis of the possible execution paths in an operation. An execution path is a sequence of actions that may be followed during the operation execution. For trivial operations (i.e. operations with neither conditional nor loop nodes) the operation contains a single execution path but, in general, several ones will exist.

To compute the execution paths we propose to represent the operation as a Model-Based Control Flow Graph (MBCFG), i.e. a control flow graph based on the model information instead on the program code, as traditional control flow graph proposals. Up to know, MBCFGs have been used to express UML sequence diagrams [18]. Here we adapt this idea to express the control flow of action-based operations.

For the sake of simplicity, when creating the MBCFG we assume that the method implementing the operation behavior is defined as a *SequenceNode* containing an ordered set of *ExecutableNodes* defining the operation effect, where each executable node may be one of the modification actions described in Section 2.3 (other types of actions are ignored during the analysis), a conditional node, a loop node or an additional sequence node (see Fig. 2.3.1). We also use two "fake" nodes, an initial node representing the first instruction in the operation and a final node representing the last one. These two nodes do not change the operation effect but help in simplifying the presentation of our MBCFG.

The digraph $MBCFG_{op}= (V_{op}, A_{op})$ for an operation op is obtained by means of the following rules:

- Every executable node in *op* is a vertex in V_{op}.
- An arc from an action vertex v_1 to v_2 is created in A_{op} if v_1 immediately precedes v_2 in an ordered sequence of nodes.
- A vertex *v* representing a conditional node *n* is linked to the vertices $v_1 \ldots v_n$ representing the first executable node for each clause (i.e. the then clause, the else clause,…) in *n*. The last vertex in each clause is linked to the vertex v_{next} immediately following *n* in the sequence of executable nodes. If *n* does not includes an else clause an arc between *v* and v_{next} is also added to A_{op}.

A vertex *v* representing a loop node *n*, is linked to the vertex representing the first executable node for *bodyPart* of *n* and to the vertex v_{next} immediately following *n* in the node sequence. The last vertex in the *bodyPart* is linked to *v* (to represent the iterative behavior).

Fig. 3.3.1 shows the MBCFG of the operations of our running example (Fig. 2.3.2). The initial and final nodes are represented as circles. Test conditions of conditional and loop nodes are not shown since they are not part of our analysis[1].

[1] Detection of infeasible paths due to unsatisfiability of tests conditions in if-then or loop nodes is out of scope of this paper. This SAT-problem can be tackled with current UML/OCL verification tools adding the test condition as an additional constraint in the model and checking if the model extended with this new constraint is still satisfiable.

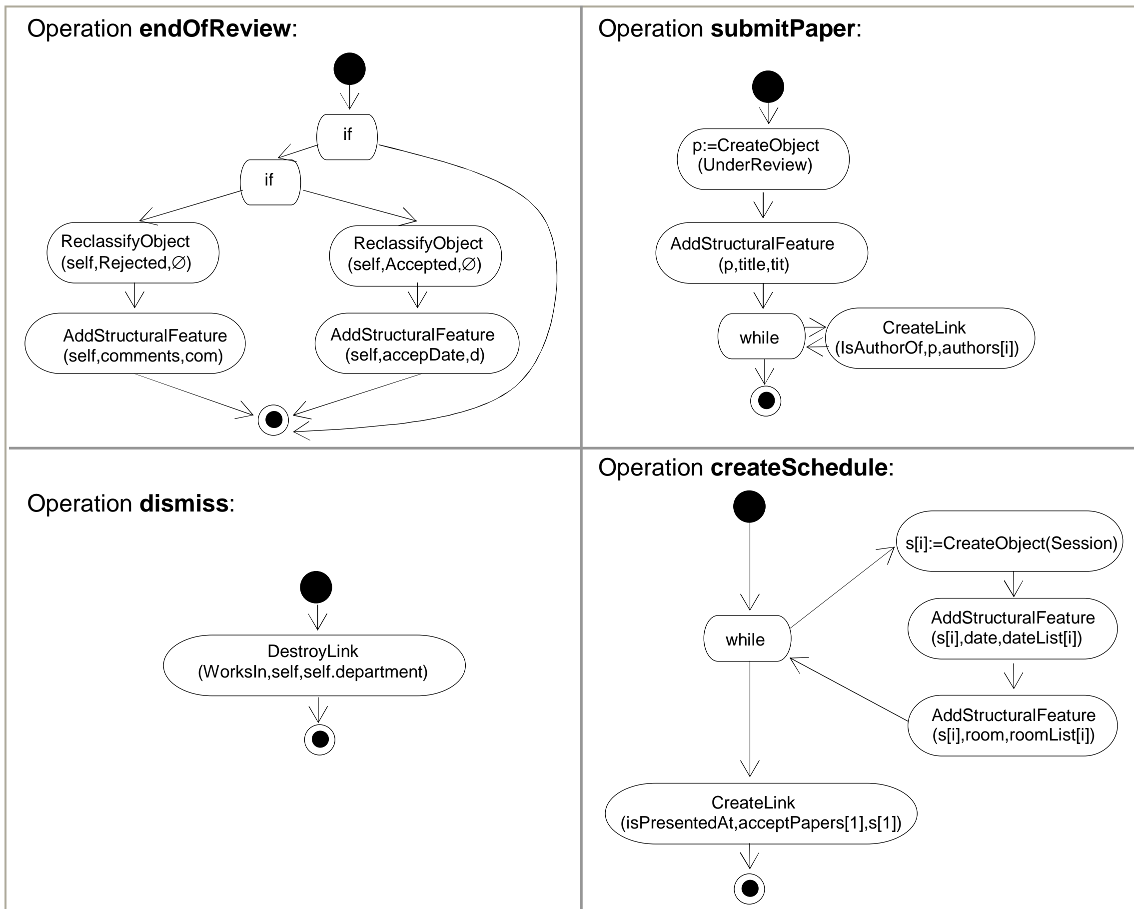


Fig. 3.3.1. MBCFG of *endOfReview, submitPaper* and *dismiss* operations for the example.

Given a $MBCFG_{op}$ graph *G*, the set of execution paths ex_{op} for *op* is defined as $ex_{op}=allPaths(MBCFG_{op})$ where *allPaths(G)* returns the set of all paths in *G* that start at the initial vertex (the vertex corresponding to the initial node), ends at the final node and does not include repeated arcs (these paths are also known as *trails* [5]).

Each path in ex_{op} is formally represented as an ordered sequence of *<number,action>* node tuples where *number* indicates the number of times that the action *action* is executed in that particular node. Vertices representing other types of executable nodes are discarded.

The number element in the tuple is only relevant for actions included in loop nodes. For other actions the number value is always '1'. For an action *action* within a loop, *number* is computed as follows:

1. Each *while-do* loop in the graph is assigned a different integer variable value *N,…,Z* representing the number of times the loop may be executed. *Do-while* loops are assigned the value *1+N,…, 1+Z* to express that the body is executed at least once.
2. The *number* of *action* is defined as the multiplication of the variable values of all loop nodes we find in the path between action and the initial vertex. That is, *action* will be executed *N* times if *action* is included in a top-level loop, *N*M* if *action* forms part of a single nested loop, and so forth.

The graphical overview of this step is:

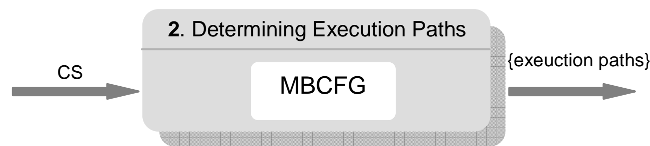


Fig. 3.3.2. Determining Execution Paths overview.

The input of the step is the CS, specifically the imperative specification of each operation included in the CS. For each operation ∈ CS, the method computes its execution path with the previous rules and returns them.

For example, Fig. 3.3.3 shows the execution paths for the MBCFG graphs shown in Fig. 3.3.1.

Operation **endOfReview**
p_1 = { }
p_2 = {<1,ReclassifyObject(self,Rejected,∅)>, <1,AddStructuralFeature(self,comments,com)>}
p_3 = {<1,ReclassifyObject(self,Accepted,∅)>, <1,AddStructuralFeature(self,accepDate,d)>}
Operation **submitPaper**
p = {<1,CreateObject(UnderReview)>, <1,AddStructuralFeature(p,title,tit)>, <N,CreateLink(IsAuthorOf,p,authors[i])>}
Operation **dismiss**
p = {<1,DestroyLink(WorksIn,self,self.department)>}
Operation **createShedule**
p = {<N,s:=CreateObject(Session)>, <N,AddStructuralFeature(s[i],date,datesList[i])>, <N,AddStructuralFeature(s[i],room,roomList[i])>, <1,CreateLink(isPresentedAt,acceptPapers[1],s[1])>}

Fig 3.3.3. Execution paths of operations of the running example.

3.4. Verifying Weak Executability

Once the execution paths of an operation have been computed, the next step of the method is to check the executability of each path.

> *An operation is **weakly executable** when there is a chance that a user may successfully execute the operation, that is, when there is at least an initial system state and a set of arguments for the operation parameters for which the execution of the actions included in the operation evolves the initial state to a new system state that satisfies all integrity constraints. Otherwise, the operation is completely useless. We define our executability property as weak executability since we do not require all executions of the operation to be successful, which could be defined as strong executability.*

As an example, consider again the operations defined for the running example of Fig. 2.1.1 and Fig. 2.3.2. Clearly, *dismiss* is not executable since every time we try to delete a link between a person *p* and a department *d* we reach an erroneous system state where *p* has no related department, a situation forbidden by the minimum

'1' cardinality constraint in the *WorksIn* relationship. In order to dismiss *p* from *d* we need to either assign a new department *d'* to *p* or to remove *p* itself within the same operation execution.

Instead, *submitPaper* is weakly executable since we are able to find an execution scenario where the new paper can be successfully submitted. Note that, classifying *submitPaper* as weakly executable does not mean that every time this operation is executed the new system state will be consistent with the constraints. For instance, if a person p passed as a value for the *authors* parameter belong to a department with already 10 submissions then the operation execution will fail because the constraint *MaxPapersSent* will not be satisfied by the system state at the end of the operation execution.

The weak executability of an operation is defined in terms of the weak executability of its execution paths: the operation is weakly executable if at least one of its paths is weakly executable[2]. Executability of a path *p* depends on the set of actions included in the path. The basic idea is the dependencies among actions, that is, some actions require the presence of other actions within the same execution path in order to leave the system in a consistent state at the end of the execution. Therefore, to be executable, a path *p* must satisfy all action dependencies for every action action in *p*. As we have seen in section 2.4, dependencies for a particular action are drawn from the structure and constraints of the class diagram and from the kind of modification performed by the action type. Following with the previous example, the *dismiss* operation is not weakly executable because its single path (see Fig. 3.3.3) is not executable since the action *DestroyLink(WorksIn,p,d)* must be always followed by a *DestroyObject(p)* or a *CreateLink(WorksIn,p,d')* to avoid violating the cardinality constraint. Since the path includes none of these actions, it is not weakly executable.

The graphical overview of this step is:

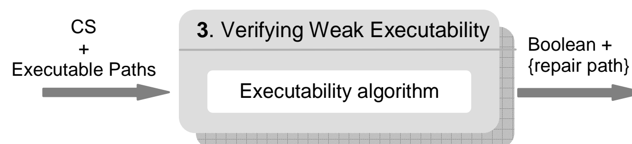


Fig. 3.4.1. Verifying Weak Executability overview.

The input of the step is the CS and the weak executable paths computed in the previous step. For each execution path, the executability algorithm verifies if it is weak executable and returns a repair path if it is not executable.

[2] It is also important to detect and repair all non-executable paths. Otherwise, all executions of the operation that follow one of those paths will irremediably fail.

3.4.1. Executability algorithm

In this section we present an algorithm for determining the weak executability of an execution path.

The specification of the function is:

Function	**weakExecutability**
Input parameters	*p* – execution path to be verified Class Diagram, composed by: set_{cl}: Classes set_{at}: Attributes (StructuralFeature) set_{as}: Associations set_{gen}: Generalizations Sets set_{constr}: Constraints
Output parameters	*reqActions* – For non-executable paths, the function returns a possible action set that must be included in the path to make it executable. Extending the path with this sequence is a necessary condition but not a sufficient one to guarantee the executability of the path. Actions in the sequence may have, in its turn, additional dependencies that must be considered as well. This can be detected by reapplying the same function over the extended path.
Result	True, if the path *p* is weak executable. False, in other case.

Firstly, the algorithm creates a copy of *p* in *pAux* (*iniPathAux* function) and adds the new auxiliary property *available* to each node. The property *node.available* keeps the number of units available for the action *node.action*. For example, the node *n = <1,0,p:=CreateObject(Person)>* from path *p* indicate that the path *p* includes an (*n.number=1*) action of type *p:=CreateObject(Person)*, but, at the moment, there are no units available (*n.available=0*). Initially, the property available takes the value of the number property (*n.available:=n.number*). This property is later used and updated by the mapping function.

Roughly, the algorithm works by iterating through the actions in the path. For each action, it computes its dependencies (*getDependencies* function). Then, the *mapping* function tries to map each dependency onto the rest of the actions in the path. Dependencies not completely satisfied are added to the *reqActions* set and return as a feedback to the user.

In the following, we present a pseudocode of the function that verifies the weak executability of an execution path:

```
function weakExecutability (in: p: Sequence (<number:Integer, action:Action>),
in: set_cl: Set(Class), in: set_at: Set (StructuralFeature), in: set_as: Set(Association),
in: set_gen: Set(GeneralizationSet), in: set_constr: Set(Constraint),
out: reqActions: Set (<number:Integer, action:Action >)): Boolean
    dependencies: Set(<number:Integer, action:Action>);
    node: <number:Integer, available:Integer, action:Action>;
    pAux: Sequence(<number:Integer, available:Integer, action:Action>);
    pAux := iniPathAux(p);
    for each node ∈ pAux do
        dependencies := getDependencies(node,set_cl,set_at,set_as,set_gen,set_constr);
        reqActions := reqActions U mapping(dependencies,pAux);
    endfor
    return (reqActions = Ø);
endfunction
```

In what follows, we explain in more detail the main steps of the algorithm and show the application to determine the weak executability of the operations of our running example.

3.4.1.1. Computing the Dependencies

As we have seen in section 2.4., the concept of dependency determines which actions are required by another action to satisfy the constraints of a class diagram.

For computing the dependencies from an action, the function *getDependencies* is founded on the dependencies table introduced in section 2.4. Given this dependency table, *getDependencies(node,*set_{cl}*,*set_{at}*,*set_{as}*,*set_{gen}*,*set_{constr}*)* function proceeds as follows:

1. Computes the set of dependencies *dep* for the action in *node.action* as stated in table 2.4.1.
2. Multiplies the *number* value in each dependency by the value of *node.number* (the number of dependencies for an action is proportional to the number of actions of that type in the path).
3. Returns *dep*.

For instance, in our running example, the call *getDependencies(<1,ur:=CreateObject(UnderReview)>)* will return the set of conjunctive actions *{<1,AddStructuralFeature(ur,title,v)>, <1,CreateLink(IsAuthorOf,ur,*o_2*)>}*, where the *v* and o_2 arguments are free variables, since they can be bounded to any object with the appropriate type.

3.4.1.2. Determining the Required Actions

For each dependency *d* returned by *getDependencies*[3], the mapping function must first check whether *d* can be mapped in one of the actions in the path and, if not, add *d* to the set of required actions *reqActions* that will be returned as a feedback to the user.

Roughly, we consider that two actions map correctly when its action type and the model elements are the same and its parameters can be bound correctly. For example, consider the path *p={<1,d:=CreateObject(Department)>,<1,AddStructuralFeature(d,name,'ComputerScience')>}* and the dependency *<1,d:=CreateObject(Department)>* → *<1,AddStructuralFeature(d,name,v)>*. In this case, the dependee action (*<1,AddStructuralFeature(d,name,v>*) can be mapped onto the second node in *p* (*<1,AddStructuralFeature(d,name,'Computer Science')>*), since the argument *v* is a free variable and it can take the value *'Computer Science'*.

It is worth to note that in some situations two or more dependee actions can be mapped (i.e. share) to the same action in the path. For example, consider the context of our running example. Suppose that we have an operation with a path *p={<N,CreateLink(IsAuthorOf,paper,authors[i]>}*. In this case, each *CreateLink(IsAuthorOf,paper,authors[i])* need a *DestroyLink(IsAuthorOf,paper,…)* or a *paper:=CreateObject(UnderReview)* (or any subclass of class *Paper*) action in the same path. The first dependency is not shareable, since each *CreateLink(IsAuthorOf,paper,…)* needs a different destroy link to keep the system consistent. Instead, the alternative dependency *paper:=CreateObject(UnderReview)* is shareable since the *N* create links may rely on the same new created object to satisfy the multiplicity dependencies.

For this purpose, we add additional column (third column) in the dependence's table to indicate if a dependency can be shareable or not.

Table 3.4.1. Table from 2.4.1 with the additional column *Shareable*.

Action	Required Actions	Shareable
o=CreateObject(c)	*AddStructuralFeature(o,at,v)* for each non-derived and mandatory attribute at of *c* or of a superclass of *c*	**No**
	AND *<min(c,as),CreateLink(as,o,o*$_2$*)>* dependencies for each non-derived association as where *c* or a superclass of *c* has mandatory participation	**No**
DestroyObject(o:c)	<min(c,as),*DestroyLink(as,o,o*$_2$*)*> for each non-derived as where *c* or a superclass of *c* has a mandatory participation	**No**
$CreateLink(as,o_1{:}c_1,o_2{:}c_2)$ (when $min(c_1,as) = max(c_1,as)$) to be repeated for the other end	$DestroyLink(as,o_1,o_3)$ (if $min(c_2,as) <> max(c_2,as)$)	**No**
	OR $CreateObject(o_1)$	**Yes**
	OR $ReclassifyObject(o_1,c_1,\emptyset)$	**Yes**
$DestroyLink(as,o_1{:}c_1,o_2{:}c_2)$ (when $min(c_1,as) = max(c_1,as)$)	$CreateLink(as,o_1,o_3)$ (if $min(c_2,as) <> max(c_2,as)$)	**No**
	OR $DestroyObject(o_1)$	**Yes**

[3] Note that, as we have seen before, *getDependencies* may return not just a single set of dependencies but several alternative ones (all of them sufficient to reach a consistent state). In those cases, the mapping function should check if at least one of the alternative dependencies sets can be successfully mapped onto the path. For the sake of simplicity this situation is not described here.

to be repeated for the other end	OR *ReclassifyObject*(o_1,Ø,c_1)	**Yes**
AddStructuralFeature(o,at,v)	-	**-**
ReclassifyObject(o,{nc},{oc})	*AddStructuralFeature(o,at,v)* for each non-derived and mandatory attribute at of each class *c* ∈ *nc*	**No**
	AND <*min(c,as),CreateLink*(*as,o,*o_3)> for each *c* ∈ *nc* and for each non-derived association *as* where *c* has a mandatory participation	**No**
	AND *ReclassifyObject(o,Ø,{cc'})* for a *cc'* such that ∃*cc* ∈ *nc* and *cc'* ∉ *oc* and *cc* ≠ *cc'* and *cc* and *cc'* are subclasses of a common disjoint and complete generalization set and *o* was instance of *cc'*	**Yes**
	AND <*min(c,as),DestroyLink*(*as,o,*o_3)> for each class *c* in *oc* and for each non-derived association *as* where *c* has a mandatory participation	**No**
	AND *ReclassifyObject(o,{cc'},Ø)* for a *cc'* such that *cc'* ∉ *nc* and ∃*cc* ∈ *oc* and *cc* ≠ *cc'* and *cc* and *cc'* are subclasses of a common disjoint and complete generalization set and *o* was instance of *cc*	**Yes**

More formally, we consider that a dependee action *d.action* can be mapped onto a node *n* in the path when the following conditions are satisfied:

1. The action type of *d.action* and *n.action* is the same (e.g. both are *CreateLink* actions).
2. The model elements modified by the actions coincide (e.g. both create new links for the association assoc).
3. All instance-level parameters of *d.action* can be bound to the parameters in n*.action* (free variables introduced by the rules may be bound to any parameter value in *n.action*, fixed ones must have the same identifier in both actions).
4. *n.number≥d.number*[4] (for dependencies that are shareable) or *n.available≥d.number* (for non-shareable actions). In this latter case, the availability value of the node is decreased according to the number of "consumed" actions: *n.available:=n.available-d.number*.

Each time a mapping for a dependency is found, if the dependency is not shareable, the available property of the mapping node is diminished (to avoid future dependencies map again to the same action) and *pAux* is updated. Dependencies that do not map with the input execution path are returned by the function *mapping* and added to the *reqActions* set.

[4] This comparison may include abstract variables (e.g. when *n* is part of a loop). In those cases, the *d.action* can be mapped iff there is a possible instantiation of the abstract variables that satisfies the inequality comparison.

3.4.1.3. Applying the algorithm

In the following, the execution of the executability function for the previous execution paths is detailed. To facilitate its understanding, free variables are shown in *italics*.

Table 3.4.2. Weak Executability for the *endOfReview* operation.

Operation: **endOfReview**	
Input	p_1 = { } pAux = { }
Output	reqActions = { } **executability = true**
Input	p_2 = {<1,ReclassifyObject(self,{Rejected},∅)>, <1,AddStructuralFeature(self,comments,com)>} pAux = {<1,1,ReclassifyObject(self,{Rejected},∅)>, <1,1,AddStructuralFeature(self,comments,com)>}
Iteration 1	node = <1,1,ReclassifyObject(self,{Rejected},∅)> dependencies = {<1,AddStructuralFeature(self,comments,*v*)>, <1,ReclassifyObject(self,Ø,{UnderReview})>} mapping: <1,AddStructuralFeature(self,comments,*v*)> maps with the second node of *pAux* <1,ReclassifyObject(self,Ø,{UnderReview})> does not map with any node of *pAux* reqActions = {<1,ReclassifyObject(self,Ø,{UnderReview})> pAux = {<1,1,ReclassifyObject(self,{Rejected},∅)>, <1,0,AddStructuralFeature(self,comments,com)>}
Iteration 2	node = <1,0,AddStructuralFeature(self,comments,com)> dependencies = { } reqActions = { <1,ReclassifyObject(self,Ø,{UnderReview})> } pAux = {<1,1,ReclassifyObject(self,{Rejected},∅)>, <1,0,AddStructuralFeature(self,comments,com)>}
Output	reqActions = {<1,ReclassifyObject(self,Ø,{UnderReview})>} **executability = false**
Input	p_3 = {<1,ReclassifyObject(self,{Accepted},∅)>, <1,AddStructuralFeature(self,accepDate,d)>} pAux = {<1,1,ReclassifyObject(self,{Accepted},∅)>, <1,1,AddStructuralFeature(self,accepDate,d)>}
Iteration 1	node = <1,1,ReclassifyObject(self,{Accepted},∅)> dependencies = {<1,AddStructuralFeature(self,accepDate,*v*)>, <1,ReclassifyObject(self,Ø,{UnderReview})>} mapping: <1,AddStructuralFeature(self,accepDate,*v*)> maps with the second node of *pAux* <1,ReclassifyObject(self,Ø,{UnderReview})> does not map with any node of *pAux* reqActions = {<1,ReclassifyObject(self,Ø,{UnderReview})>} pAux = {<1,1,ReclassifyObject(self,{Accepted},∅)>, <1,0,AddStructuralFeature(self,accepDate,d)>}
Iteration 2	node = <1,0,AddStructuralFeature(self,accepDate,d)> dependencies = { } reqActions = { <1,ReclassifyObject(self,Ø,{UnderReview})> } pAux = { <1,1,ReclassifyObject(self,Accepted,∅)>, <1,0,AddStructuralFeature(self,accepDate,d)>}
Output	reqActions = {<1,ReclassifyObject(self,Ø,{UnderReview})>} **executability = false**

The execution path p_1 is weakly executable, since it does not contain any action. On the other hand, the execution paths p_2 and p_3 are not weakly executable since they always violate the disjointness constraint of the generalization. The action required to make the paths executable is *ReclassifyObjectAction(self,Ø,{UnderReview})* in both cases.

Table 3.4.3. Weak Executability for the *submitPaper* operation.

Operation: **submitPaper**	
Input	p = {<1,p:=CreateObject(UnderReview)>, <1,AddStructuralFeature(p,title,tit)>, <N,CreateLink(IsAuthorOf,(p,authors[i])>} pAux = {<1,1,p:=CreateObject(UnderReview)>, <1,1,AddStructuralFeature(p,title,tit)>, <N,N,CreateLink(IsAuthorOf,p,authors[i])>}
Iteration 1	node = <1,1,p:=CreateObject(UnderReview)> dependencies = {<1,AddStructuralFeature(p,title,*v*)>, <1,CreateLink(IsAuthorOf,p,o_2)>} mapping: <1,AddStructuralFeature(p,title,*v*)> maps with the second node of *pAux* <1,CreateLink(IsAuthorOf,p,o_2)> maps with the third node of *pAux* reqActions = { } pAux = {<1,1,p:=CreateObject(UnderReview)>, <1,0,AddStructuralFeature(p,title,tit)>, <N,N-1,CreateLink(IsAuthorOf,(p,authors[i])>}
Iteration 2	node = <1,0,AddStructuralFeature(p,title,tit)> dependencies = { } reqActions = { } pAux = {<1,1,p:=CreateObject(UnderReview)>, <1,0,AddStructuralFeature(p,title,tit)>, <N,N-1,CreateLink(IsAuthorOf,(p,authors[i])>}
Iteration 3	node = <N,N-1,CreateLink(IsAuthorOf,(p,authors[i])> dependencies = { <N,DestroyLink(IsAuthorOf,p,o_3)> OR <N, *o*:=CreateObject(Paper)> } mapping: <N,DestroyLink(IsAuthorOf,p,o_3)> does not map with any action of *pAux*. <N,*o*:=CreateObject(Paper)> maps with the first node of *pAux*, since this dependence is shareable and UnderReview is a subclass of Paper. reqActions = { } pAux = {<1,1,p:=CreateObject(UnderReview)>, <1,0,AddStructuralFeature(p,title,tit)>, <N,N-1,CreateLink(IsAuthorOf,(p,authors[i])>}
Output	reqActions = { } **executability = true**

The execution path *p* is weakly executable, since all actions required are contained in the execution path.

Table 3.4.4. Weak Executability for the *dismiss* operation.

Operation: **dismiss**	
Input	p = {<1,DestroyLink(WorksIn,self,self.department)>} pAux = {<1,1,DestroyLink(WorksIn,self,self.department)>}
Iteration 1	node = <1,1,DestroyLink(WorksIn,self,self.department)> dependencies = { <1,CreateLink(WorksIn,self,o_2)> OR <1,DestroyObject(self)> } mapping: <1,CreateLink(WorksIn,self,o_2)> does not map with any node of *pAux* <1,DestroyObject(self)> does not map with any node of *pAux* reqActions = { <1,CreateLink(WorksIn,self,o_2)> OR <1,DestroyObject(self)> } pAux = {<1,1,DestroyLink(WorksIn,self,self.department)>}
Output	reqActions = { <1,CreateLink(WorksIn,self,o_2)> OR <1,DestroyObject(self)> } **executability = false**

This execution path is not executable (and thus, neither the dismiss operation since this is its only path), because removing the link violates the cardinality constraint ‘1’ of *WorksIn* association. Adding the required actions *CreateLink(WorksIn,self,o_2)* (i.e. adding a new link for the dangling object) OR *DestroyObject(self)* (i.e. destroying it) returned by our method would make the path executable.

Table 3.4.5. Weak Executability for the *createShedule* operation.

Operation: **createSchedule**	
Input	p = {<N,s:=CreateObject(Session)>, <N,AddStructuralFeature(s,date,dateList[i])>, <N,AddStructuralFeature(s,room,roomList[i])>, <1,CreateLink(isPresentedAt,acceptPapers[1],s[1])>} pAux = {<N,N,s:=CreateObject(Session)>, <N,N,AddStructuralFeature(s,date,dateList[i])>, <N,N,AddStructuralFeature(s,room,roomList[i])>, <1,1,CreateLink(isPresentedAt,acceptPapers[1],s[1])>}
Iteration 1	node = <N,s:=CreateObject(Session)> dependencies = {<N,AddStructuralFeature(s,date,v_1)>, <N,AddStructuralFeature(s,room,v_2)>, <N,CreateLink(isPresentedAt,v_3,s[i])> } mapping: <N,AddStructuralFeature(s,date,v_1)> maps with the second node of *pAux* <N,AddStructuralFeature(s,room,v_2)> maps with the third node of *pAux* <1,CreateLink(isPresentedAt,v_3,s[i])> maps with the fourth node of *pAux* and <N-1, CreateLink(isPresentedAt,v_3,s[i])> cannot map with any node due to the fault of units reqActions = { <N-1,CreateLink(isPresentedAt,v_3,s[i])> } pAux = {<N,N,s:=CreateObject(Session)>, <N,0,AddStructuralFeature(s,date,dateList[i])>, <N,0,AddStructuralFeature(s,room,roomList[i])>, <1,0,CreateLink(isPresentedAt,acceptPapers[1],s[1])>}
Iteration 2	node = <N,0,AddStructuralFeature(s,date,datesList[i])> dependencies = { } reqActions = { <N-1,CreateLink(isPresentedAt,v_3,s[i])> } pAux = {<N,N,s:=CreateObject(Session)>, <N,0,AddStructuralFeature(s,date,dateList[i])>, <N,0,AddStructuralFeature(s,room,roomList[i])>, <1,0,CreateLink(isPresentedAt,acceptPapers[1],s[1])>}
Iteration 3	node = <N,0,AddStructuralFeature(s,date,roomList[i])> dependencies = { } reqActions = { <N-1,CreateLink(isPresentedAt,v_3,s[i])> } pAux = {<N,N,s:=CreateObject(Session)>, <N, 0, AddStructuralFeature(s,date,dateList[i])>, <N,0,AddStructuralFeature(s,room,roomList[i])>, <1,0,CreateLink(isPresentedAt,acceptPapers[1],s[1])>}
Iteration 4	node = <1, 0, CreateLink(isPresentedAt,acceptPapers[1],s[1])> dependencies = { } reqActions = { <N-1,CreateLink(isPresentedAt,v_3,s[i])> } pAux = {<N,N,s:=CreateObject(Session)>, <N,0,AddStructuralFeature(s,date,dateList[i])>, <N,0,AddStructuralFeature(s,room,roomList[i])>, <1,0,CreateLink(isPresentedAt,acceptPapers[1],s[1])>}
Output	reqActions = { <N-1,CreateLink(isPresentedAt,v_3,s[i])> } **executability = false**

The execution path *p* is not weakly executable since it always violates the cardinality constraint ‘1’ of paper role in *IsPresentedAt* link. The action required to make the path executable is one *CreateLink(isPresentedAt,v_3,s[i])* action for each session created, where v_3 represents an accepted paper. Note that this set of actions can be included inside the loop.

3.5. Verifying Completeness

Users evolve the system state by executing the set of operations defined in the class diagram.

> *We consider that an operation set is* ***complete*** *when all possible changes (inserts/updates/deletes/...) on the system state can be performed through the execution of the operations defined in the class diagram. Otherwise, there will be parts of the system that users will not be able to modify since no available operations address their modification.*

For instance, the set of operations defined in Fig. 2.3.2 is incomplete since operations to remove a person or to create and remove departments are not specified, among others.

More formally, an operation set $set_{op}=\{op_1,\ldots, op_n\}$ is complete when for each modifiable element *e* in the class diagram and each possible action *action* modifying the population of *e*, there is at least a weak executable path in some op_i that includes *action*.

The graphical overview of this step is:

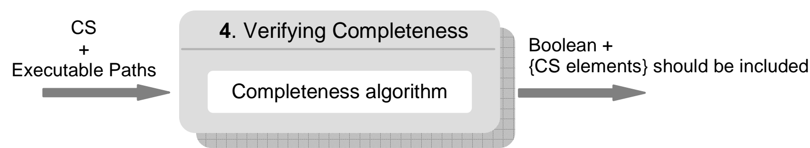


Fig. 3.5.1. Verifying Completeness overview.

The input of the step is the CS and the weak executable paths computed in the step 2. For each modifiable element e defined in the CS, the completeness algorithm checks if exists an action included in some execution path that can modify the state of e and returns as a feedback the elements that cannot be modified.

3.5.1. Completeness algorithm

In this section we present an algorithm for determining the completeness of a set of operations.

The specification of the function is:

Function	**Completeness**
Input parameters	CS, composed by: set_{cl}: Classes set_{at}: Attributes (StructuralFeature) set_{as}: Associations set_{gen}: Generalizations Sets set_{op}: Operations *exPaths*: set of execution paths from set_{op}
Output parameters	*feedback* – For incomplete operations sets, the output parameter feedback contains the set of actions that should be included in some operation to satisfy the completeness property.
Result	True, if the set of operations is complete. False, in other case.

Roughly, the algorithm proceeds as follows:

1. Obtains all different actions of weak executable paths of the set_{op} operations set (*getExistingActions* function).
2. Computes the set of actions that should be provided to the system users in order to be able to modify all parts of the system state, depending on the structure and properties of the class diagram (*getRequiredActions* function). Next subsection provides the rules for determining such actions.
3. Subtracts the existing actions from required actions (*requiredActionsSet – existingActionsSet = feedbak*) to obtain the actions that should be include. Note that, if the result is an empty set, then the set_{op} is complete.

A function for checking the completeness of setop is the following:

```
function completeness (in: set_cl: Set(Class), in: set_at: Set (StructuralFeature), in: set_as: Set(Association),
in: set_gen: Set(GeneralizationSet), in: set_op: Set (Operations),
in: exPaths: Set (Sequence (<number:Integer, action:Action>)), out: feedback: Set(Action)): Boolean
    requiredActionsSet, existingActionsSet: Set(Action);
    action: Action;
    feedback := Ø;
    existingActionsSet := getExistingActions(exPaths);
    requiredActionsSet := getRequiredActions(set_cl, set_at, set_as, set_gen);
    for each action ∈ requiredActionsSet do
        if (action ∉ existingActionsSet) then feedback := feedback U {action}; endif
    endfor
    return (feedback = Ø);
endfunction
```

In what follows, we explain in more detail the main steps of the algorithm and show the application to determine the weak of the operations of our running example.

3.5.1.1. Computing the Existing Actions

The set of actions returned by *getExistingActions* is composed by those actions that are included in some weak executable path of the set *exPaths*.

3.5.1.2. Computing the Required Actions

The set of actions returned by *getRequiredActions* is computed by first determining the modifiable model elements in the class diagram (i.e. the elements whose value or population can be changed by the user at run-time) and then deciding, for each modifiable element, the possible types of actions that can be applied on it.

The modifiable elements can be summarized in:

- **Class**: A class is modifiable as long as it is not an abstract class and it is not the supertype of a complete generalization set (instances of such supertypes must be created/deleted through their

subclasses). For each modifiable class *c*, users must be provided with the actions *CreateObject(c)* and *DestroyObject(o:c)*[5] to create and remove objects from *c*.

- **Attribute** (*StructuralFeature*): An attribute is modifiable when it is not derived[6]. For each modifiable attribute *attr* the action *AddStructuralFeature(o,attr,v)* is necessary.
- **Association**: An association is modifiable if all its member ends are not derived. For each modifiable association *assoc*, we need the actions *CreateLink(assoc,*p_1*,*p_2*)* and *DestroyLink(asso,*p_1*,*p_2*)*.
- **Generalization**: Generalization sets are always modifiable. For generalizations sets we need a set of actions *ReclassifyObject(o,{nc},{oc})* among the classes involved in the generalization to specialize (generalize) the object *o* to (from) each subclass of the generalization.

3.5.1.3. Applying the algorithm

In the following, the execution of the completeness function for our running example (Fig. 2.1.1 and 2.3.2) is detailed.

The operation *getExistingActions* retrieves all different actions of weak executable paths of the operations *endOfReview*, *submitPaper*, *dismiss* and *createShedule*:

existingActionsSet = { *ur*:=CreateObject(UnderReview), AddStructuralFeature(*ur*,title,t), CreateLink(IsAuthorOf,*p*,*ur*) }

The operation getRequiredActions returns the following set of actions (free variables are shown in *italics*):

requiredActionsSet = {

//One CreateObject(Class) action for each modifiable class of the diagram:

a:=CreateObject(Accepted), *r*:=CreateObject(Rejected), *ur*:=CreateObject(UnderReview), *p*:=CreateObject(Person), *d*:=CreateObject(Department), *s*:=CreateObject(Session),

//One DestroyObject(obj) action for each modifiable object of the diagram:

DestroyObject(*a*), DestroyObject(*r*), DestroyObject(*ur*), DestroyObject(*p*), DestroyObject(*d*), DestroyObject(*s*),

//One AddStructuralFeature(obj,att,v) action for each modifiable attribute att:

AddStructuralFeature(*a*,title,t), AddStructuralFeature(*a*,accepDate,d), AddStructuralFeature(*r*,comments,c), AddStructuralFeature(*p*,name,n), AddStructuralFeature(*p*,email,e), AddStructuralFeature(*d*,name,n), AddStructuralFeature(*s*,date,d), AddStructuralFeature(*s*,room,s),

//One CreateLink(as,p_1*,*p_2*) action for each modifiable association as:*

CreateLink(IsAuthorOf,*p*,*a*), CreateLink(WorksIn,*p*,*d*), CreateLink(isPresentedAt,*a*,*s*),

//One *DestroyLink(as,*p_1*,*p_2*)* action for each modifiable association *as*

DestroyLink(IsAuthorOf,*p*,*a*), DestroyLink(WorksIn,*p*,*d*), DestroyLink(IsPresentedAt,*a*,*s*),

//One ReclassifyObject(o,nc,oc) for each classes involved in the generalization to specialize (generalize) the object o to (from) each subclass of the generalization:

ReclassifyObject(*ur*,{Accepted},{UnderReview}), ReclassifyObject(*ur*,{Rejected},{UnderReview}), ReclassifyObject(*a*,{UnderReview},{Accepted}), ReclassifyObject(*a*,{Rejected},{Accepted}), ReclassifyObject(*r*,{UnderReview},{Rejected}), ReclassifyObject(*r*,{Accepted},{Rejected}) }

---

[5] Or a generic operation *DestroyObject(o:OclAny)* to remove objects of any class.

[6] Read-only attributes are considered modifiable because users must be able to initialize their value (and similar for read-only associations).

Therefore, the output parameter *feedback* contains the set of actions that should be included in some operation to satisfy the completeness property.

feedback = {

a:=CreateObject(Accepted), *r*:=CreateObject(Rejected), *p*:=CreateObject(Person), *d*:=CreateObject(Department), *s*:=CreateObject(Session), DestroyObject(*a*), DestroyObject(*r*), DestroyObject(*ur*), DestroyObject(*p*), DestroyObject(*d*), DestroyObject(*s*),
AddStructuralFeature(*a*,accepDate,d), AddStructuralFeature(*r*,comments,c), AddStructuralFeature(*p*,name,n), AddStructuralFeature(*p*,email,e), AddStructuralFeature(*d*,name,n), AddStructuralFeature(*s*,date,d), AddStructuralFeature(*s*,room,s), CreateLink(WorksIn,*p*,*d*), CreateLink(isPresentedAt,*a*,*s*), DestroyLink(IsAuthorOf,*p*,*a*), DestroyLink(WorksIn,*p*,*d*), DestroyLink(IsPresentedAt,*a*,*s*), ReclassifyObject(*ur*,{Accepted}, {UnderReview}), ReclassifyObject(*ur*, {Rejected}, {UnderReview}), ReclassifyObject(*a*,{UnderReview}, {Accepted}), ReclassifyObject(*a*, {Rejected}, {Accepted}), ReclassifyObject(*r*,{UnderReview}, {Rejected}), ReclassifyObject(*r*, {Accepted}, {Rejected}) } }

3.6. Detecting Redundant Paths

The last step of our method consists in detect redundant paths.

> *An action (or set of actions) in an execution path is* ***redundant*** *if its effect on the system state is subsumed by the effect of later actions in the same path, that is, the final system state when executing the operation following that path would be exactly the same with or without the redundant action.*

The aim of this step is detect the redundant actions and inform the designer so that they can remove it from the path.

The graphical overview of this step would be:

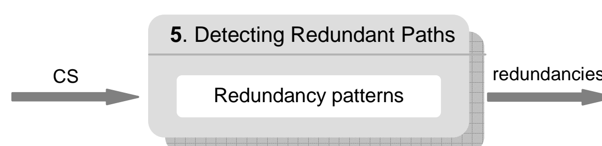


Fig. 3.6.1. Detecting Redundant Paths overview.

The input of the step is the CS, specifically the imperative specification of each operation included in the CS. For each set of actions, the method would detect if it follow some redundancy pattern and would return as a feedback the actions (or set of actions) that are redundant.

An operation specification may be redundant at three different levels:

1. Some actions in an execution path are redundant.
2. The complete execution path is redundant.
3. The operation as a whole is itself redundant.

3.6.1. Redundancy in Actions

We have identified several patterns that detect such redundant actions. For each pattern we provide a possible non-redundant alternative path. However, the modification of the paths cannot be fully automatic since, for instance, a redundant action *action* may not be redundant in a different path also including action or may affect the execution of other actions in the path. Nevertheless, we believe it is worth to at least point out these redundant actions to the designer.

Table 3.6.1. Patterns of redundant paths.

Redundant path	Equivalent path	Redundancy
{…, AddStructuralFeatureValue(o,at,v), …, AddStructuralFeatureValue(o,at,v_2), …}	{… AddStructuralFeatureValue(o,at,v_2) …}	The second update overwrites the first one
{…, o = CreateObject(Cl), …, DeleteObject(o), …}	{…}	No need of creating/updating/reclassifying an object that it is going to be removed within the same execution.
{…, AddStructuralFeatureValue(o,at,v), …, DeleteObject(o), …}	{…}	
{…, ReclassifyObject(o,{nc},{oc}), …, DeleteObject(o), …}	{…}	
{…, CreateLink(as,p_1,p_2) …, DeleteLink(as,p_1,p_2), …}	{…}	Why creating a link that it is going to be removed?
{…, ReclassifyObject(o,Cl_2,Cl_1), …, ReclassifyObjectAction(o,Cl_1,Cl_2), …}	{… ReclassifyObjectAction(o,Cl_2,Cl) …}	The last reclassification removes the effect of the first one.
{…, ReclassifyObject(o,Cl_2,Cl_1), …, ReclassifyObjectAction(o,Cl_3,Cl_2), …}	{…, ReclassifyObject (o,Cl_3,Cl_1),…}	Transitive property

As an illustrative example of this step, suppose that we have a new operation in our running example:

```
context UnderReview::removePaper() {
  ReclassifyObject(self,{Rejected},{UnderReview});
  DeleteObject(self);
}
```

As we can see in the fourth pattern, the previous operation includes a redundant action, *ReclassifyObject(self,{Rejected},{UnderReview})*, because it is redundant reclassifying an object that it is going to be removed within the same execution.

3.6.2. Redundancy in Execution Paths

An execution path p_1 is redundant with respect to an execution path p_2 (of the same or a different operation) when p_1 is subsumed by p_2, i.e. when all actions in p_1 appear in p_2 with the same or lower number. This may be perfectly correct (e.g. p_1 may appear in a basic operation whose behavior is also included in a more complex one) but it should be highlighted as suspicious, specially when it happens also that p_2 is redundant respect to p_1, meaning that both paths have exactly the same actions.

As an illustrative example of this step, suppose that we have the follow new operations in our running example:

```
context Paper::setTitle(tit:String) {
  if (self.oclIsTypeOf(UnderReview)) AddStructuralFeature(self,title,tit);
  else
     if (self.oclIsTypeOf(Accepted)) AddStructuralFeature(self,title,tit);
     else
        if (self.oclIsTypeOf(Rejected)) AddStructuralFeature(self,title,tit);
     endif
  endif
}
```

Obviously, two of the three execution paths are redundant, since all execution paths modify the same element with the same value.

3.6.3. Redundancy in Operations

We say that an operation *op* may be redundant when all its execution paths are redundant, especially when all its paths can be mapped to the paths of the other operation op_2. Even if both operations make sense, designer could probably merge them to favor the simplicity of the schema.

As an illustrative example of this step, suppose that we have the follow new operations in our running example:

```
context Paper::setTitle(tit:String) {
  AddStructuralFeature(self,title,tit);
}
contextPaper::createPaperUnderReview(tit:String) {
  p:Paper;
  p := CreateObject(Paper);
  AddStructuralFeature(self,title,tit);
  ReclassifyObject(self,{UnderReview},{})
}
```

In this case, the operation *setTitle* may be redundant, since its execution path can be mapped to the execution path of *createPaperUnderReview*.

4. Related Work

The properties that we verify in this work have been studied previously in more or less depth. In the follow, we summarize the main related works for each property.

SYNTACTIC CONSISTENCY

The syntactic consistency of UML artifacts has been studied in several works. [35], for example, defines four constraints that must be checked in order to guarantee that a dynamic diagram is consistent with a class diagram. [13] defines a set of consistency rules for validate a UML model. [42] uses the Description Logics formalism [12] for maintaining consistency between (evolving) UML models. Our method complements these purposes adding new rules at metamodel level.

EXECUTABILITY

In general, the verification of UML specifications is done through two steps: (1) translating the UML specification in a specific formal language and (2) verifying the obtained formal specification by means of a suitable technique.

The formal language and the technique used depend on the properties we want to verify. For verifying dynamic properties (like executability) a technique widely used is *Model Checking*.

Model Checking is an approach emerged for verifying requirements, mainly in developing reliable software for concurrent systems. The essential idea behind model checking is shown in Fig. 4.1. A Model-Checking tool accepts system requirements or designs (called models) and a dynamic property (called specification) that the final system is expected to satisfy. The tool then outputs yes if the given model satisfies the given specifications and generates a counterexample otherwise. The counterexample details why the model does not satisfy the specification. By studying the counterexample, you can pinpoint the source of the error in the model, correct the model, and try again. The idea is that by ensuring that the model satisfies enough system properties, we increase our confidence in the correctness of the model.

Roughly, model checkers work by generating and analyzing all the potential operation executions at run-time and evaluating if for each (or some) execution the given property is satisfied. Even though a number of optimizations are available in state-of-the-art model checkers (as partial order reduction, state compression, abstraction and so forth), verification methods based on model checking suffer from the state explosion problem (i.e. the size of the problem grows exponentially in terms of the size of the model and thus, in general it is not possible to explore all possible executions). This implies that usually results provided by these methods are not conclusive, i.e. absence of a solution cannot be used as a proof, that is, an operation classified as not weakly executable may still have a correct execution outside the search space explored during the verification. Instead, our analysis is static and thus no animation/simulation of the model is required.

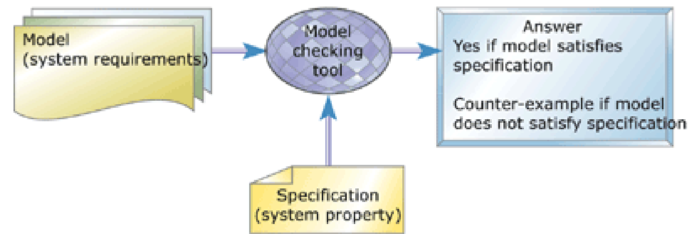


Fig. 4.1. Model-Checking overview.

There are a variety of tools that implements Model Checking technique. One of the most popular and open-source is the SPIN Model Checker [20]. SPIN supports a formal language to specify systems descriptions, called PROMELA (a PROcess MEta LAnguage). Another Model Checking-based tool is ProB [38], that supports the B-Method [1].

Model Checking approach has been used in the context of behavior UML specifications, mainly in the verification of state machines [23, 22, 28], activity diagrams [16] and on the consistent interrelationship between them and/or the class diagram [21, 11, 19, 43].

As we have seen, to check the executability of an operation (or, in general, any property that can be expressed as a Linear Temporal Logic formula – LTL [14]) previous works rely on the use of Model Checking techniques. Many of these works restrict the expressivity of the supported UML models. In fact, most of the methods above do not accept the specification of actions in the input behavior specifications, which is exactly the focus of our method. A remarkable difference of our method is that, since do not require animation/simulation and do not restrict the language, is efficient and complete: the existence of a solution can always be determined.

As a trade-off our method is unable to verify arbitrary temporal properties. We believe our method could be used to perform a first correctness analysis, basic to ensure a minimum quality level in the operation specification. Then, designers could complement the verifying process proceeding with a more detailed analysis adapting current approaches presented above to the verification of operations specified with action semantics. For instance, example execution traces that make the operation reach an error state would help designers to detect particular scenarios not yet appropriately considered in the operation.

Moreover, there is also a difference in the kind of feedback provided to the designer when the executability property is not satisfied. Model checking tools are able to provide example execution traces that do not satisfy the integrity constraints. In contrast, our method provides a more valuable feedback (at least for a first-level correctness analysis) since it suggests how to change the operation specification in order to repair the detected inconsistency.

COMPLETENESS

Regarding this property, we would like to remark that, to the best of our knowledge, our work is the first one proposing the verification of the completeness property of a behavioral schema.

REDUNDANCY

Redundant property has been studied in several perspectives. [10] identify redundancies between Sequence Diagrams and declarative contracts of operations. [4] computes the net effect, that is, defines structural inconsistencies (redundancies) between construction operations (actions), similar to our detection of redundancies in actions.

5. Conclusions and Further Work

We have presented an efficient and decidable method for verifying the correctness of imperative operations specified using the action semantics formalism, one of the key elements in all MDD and UML Executable methods. It is worth to note that our method only treats a subset of the actions provided by UML, but this set can be extended to tackle the whole range of actions.

Our method is able to verify several properties of the behavior specifications: syntactic consistency, executability, completeness and redundancy. All the process is based on a static analysis of the structural and behavioral schemas and, for verify the executability, is also based on the dependencies among the actions included in the operation specification. For verify the executability model animation/simulation is not performed during the verification process, and thus, our method does not suffer from the state-explosion problem of current model-checking based methods [9]. As a trade-off, our method is not adequate for evaluating general LTL properties.

We believe that the characteristics of our method make it especially suitable for its integration in current CASE and code-generation tools, as part of the default consistency checks that those tools should continuously perform to assist designers in the definition of software models. Moreover, the valuable feedback provided by our method helps designers to correct the detected errors since our method is able to suggest a possible repair procedure instead of just highlighting the problem.

As a further work, we plan to extend the set of actions our method deals with and to apply our techniques to other kinds of UML behavior specifications, as state machines and interactions, that may also include action sequences (for instance, as part of a state transition).

Moreover, we would like to complement our techniques by providing an automatic transformation between the initial action semantics specification and the input language of a popular model-checker tool (as the PROMELA language [20] so that, after an initial verification with our techniques, designers may get a more fine-grained (though partial) analysis by means of applying model checking techniques on the operation specification.

6. References

1. Abrial, J. R.: *The B-Book: Assigning Programs to Meanings*. Cambridge University Press. (1996)
2. Atkinson, C., Kühne, T.: *Model-Driven Development: A Metamodeling Foundation*. IEEE Software 20(5), 36-41 (2003)
3. Bauerdick, H., Gogolla, M., Gutsche, F.: *Detecting OCL Traps in the UML 2.0 Superstructure: An Experience Report*. Int. Conf. on the Unified Modeling Language, LNCS, 3273, 188-197 (2004)
4. Blanc, X., Mougenot, A.: *Detecting Model Inconsistency through Operation-Based Model Construction*. Int. Conf. on Soft. Eng., 511-520, (2008)
5. Bollobas, B.: *Modern graph theory*. Springer (2002)
6. Cabot, J.: *From Declarative to Imperative UML/OCL Operation Specifications*. LNCS, 4801, 198-213 (2007)
7. Cabot, J., Gómez, C.: *Deriving Operation Contracts from UML Class Diagrams*. Int. Conf. on Model Driven Engineering Languages and Systems, LNCS, 4735, 196-210, (2007)
8. Raistrick C., Francis P., Wright J.: *Model driven architecture with executable UML*. Chapter 10 (2004)
9. Clarke, E. M., Grumberg, O., Jha, S., Lu, Y., Veith, H: *Progress on the State Explosion Problem in Model Checking*. Informatics - 10 Years Back. 10 Years Ahead. R. Wilhelm, LNCS, 176-194 (2001)
10. Costal, D., Sancho, M. R., Teniente, E.: *Understanding Redundancy in UML Models for Object-Oriented Analysis*. Int. Conf. on Advanced Information Systems Engineering, 659-674 (2002)
11. Gallardo, M.M., Merino, P., Pimentel, E.: *Debugging UML Designs with Model Checking*. Journal of Object Technology, 1(2), 101-117 (2002)
12. Donini, F. M., Lenzerini, M., Nardi, D., Schaerf, A.: *Reasoning in Description Logics*. Principles of Knowledge Representation, 191–236 (1996)
13. Egyed, A.: *Instant Consistency Checking for the UML*. Int. Conf. on Soft. Eng., 381-390 (2006)
14. Emerson, E. A.: *Temporal and Modal Logic*. Handbook of Theoretical Computer Science, 8, 995-1072 (1990)
15. Engels, G., Küster, J. M., Heckel, R., Groenewegen L.: A *Methodology for Specifying and Analyzing Consistency of Object-Oriented Behavioral Models*. ACM SIGSOFT Software Engineering Notes, 26, 186-195 (2001)
16. Eshuis, R.: *Symbolic Model Checking of UML Activity Diagrams*. ACM Transactions on Soft. Eng. and Methodology, 15, 1-38 (2006)
17. France, R. B., Ghosh, S., Dinh-Trong, T., Solberg, A.: *Model-Driven Development using UML 2.0: Promises and Pitfalls*. COMPUTER, 59-66, (2006)
18. Garousi, V., Briand, L., Labiche, Y.: *Control Flow Analysis of UML 2.0 Sequence Diagrams*. European Conf. on Model Driven Architecture-Foundations and Applications, LNCS, 3748, 160-174 (2005)
19. Graw, G., Herrmann, P.: *Transformation and Verification of Executable UML Models*. Electronic Notes in Theoretical Computer Science, 101, 3-24 (2004)

20. Holzmann, G. J.: *The spin model checker: Primer and reference manual*. Addison-Wesley Professional (2004)
21. Knapp, A., Wuttke, J.: *Model Checking of UML 2.0 Interactions*. Workshop on Critical Systems Development using Modelling Languages, LNCS, 4364, 42-51 (2006)
22. Latella, D., Majzik, I., Massink, M.: *Automatic Verification of a Behavioural Subset of UML Statechart Diagrams using the SPIN Model-Checker*. Formal Aspects of Computing, 11(6), 637-664 (1999)
23. Lilius, J., Paltor, I. P.: *Formalising UML State Machines for Model Checking*. Int. Conf. on the Unified Modeling Language, LNCS, 1723, 430–445 (1999)
24. McAllister, A. J., Sharpe, D.: *An Approach for Decomposing N-Ary Data Relationships*. Software-Practice & Experience, 28(1), 125-154 (1998)
25. Mellor Stephen J., Balcer Marc J.: *Executable UML: A foundation for model-driven architecture*. Addison-Wesley (2002)
26. Mellor, S. J., Scott, K., Uhl, A., Weise, D.: *Model-Driven Architecture*. Computing Reviews, 45, 631 (2004)
27. Meyer, B.*: Applying 'Design by Contract'*. Computer, 25, 40-51 (1992)
28. Ober, I., Graf, S., Ober, I.: *Validating Timed UML Models by Simulation and Verification*. Int. Journal on Software Tools for Technology Transfer, 8(2), 128-145 (2006)
29. Object Action Language (OAL). *http://66.102.1.104/scholar?hl=ca&lr=&q=cache:1gQDIRCqzUIJ:www.acceleratedtechnology.com/pdf_download/bpalref.pdf+%22Object+Action+Language%22*
30. Object Management Group (OMG): *UML 2.0 Superstructure Specification*. OMG Adopted Specification (ptc/07-11-02) (2007)
31. Object Management Group (OMG): *Object Constraint Language Specification* (formal/2006-05-01).
32. Object Management Group (OMG): *UML ASL Reference Guide*. *www.omg.org/docs/ad/03-03-12.pdf*
33. Olivé, A.: *Conceptual modeling of information systems*. Springer (2007)
34. Olivé, A., Cabot, J.: *A Research Agenda for Conceptual Schema-Centric Development*. Conceptual Modelling in Information Systems Engineering, 319-334 (2007)
35. Paige, R. F., Brooke, P. J., Ostroff, J. S.: *Metamodel-Based Model Conformance and Multiview Consistency Checking*. ACM Transactions on Soft. Eng. and Methodology, 16(3) (2007)
36. PathMATE Action Language (PMAL). *http://www.pathfindermda.com/products/spotlight.php*
37. Platform-independent Action Language (PAL). *ftp://ftp.software.ibm.com/software/rational/web/casestudy/RAC14010-USEN-00.pdf*
38. ProB. *http://users.ecs.soton.ac.uk/mal/systems/prob.html*
39. Rumbaugh, J., Jacobson, I., Booch, G.: *The unified modeling language reference manual*. Addison-Wesley Professional; Har/Cdr edition (1999)
40. Shlaer-Mellor Action Language (SMALL). *http://www.modelint.com/downloads/small.pdf*
41. Starr's Concise Relational Action Language (SCRALL). *http://www.modelint.com/downloads/mint.scrall.tn.1.pdf*
42. Van Der Straeten, R., Mens, T., Simmonds, J., Jonckers, V.: *Using Description Logic to Maintain Consistency between UML Models*. Int. Conf. on the Unified Modeling Language, LNCS, 2863, 326–340 (2003)
43. Xie, F., Levin, V., Browne, J. C.: *Model Checking for an Executable Subset of UML*. Int. Conf. on Automated Soft. Eng, 333-336 (2001)